Conceptual Modeling of Hybrid Polystores

Gobert, Maxime; Meurice, Loup; Cleve, Anthony

Published in:
Proceedings of the 40th International Conference on Conceptual Modeling (ER 2021)

Publication date:
2021

Document Version
Peer reviewed version

Conceptual Modeling of Hybrid Polystores

Maxime Gobert ✉, Loup Meurice, and Anthony Cleve

Namur Digital Institute, University of Namur, Belgium
{firstname.lastname}@unamur.be

Abstract. An increasing number of organisations rely on NoSQL technologies to manage their mission-critical data. However, those technologies were not intended to *replace* relational database management systems, but rather to *complement* them. Hence the recent emergence of heterogeneous database architectures, commonly called *hybrid polystores*, that rely on a *combination* of several, possibly overlapping relational and NoSQL databases. Unfortunately, there is still a lack of models, methods and tools for data modeling and manipulation in such architectures. With the aim to fill this gap, we introduce HyDRa, a conceptual framework to design and manipulate hybrid polystores. We present the HyDRa textual modeling language allowing one to specify (1) the conceptual schema of a polystore, (2) the physical schemas of each of its databases, and (3) a set of mapping rules to express possibly complex correspondences between the conceptual schema elements and the physical databases.

Keywords: Hybrid polystores · Conceptual modeling · Framework

1 Introduction

NoSQL technologies have been around for more than a decade, and a large number of organisations are currently using them to store and manipulate mission-critical data. Despite their increasing popularity it becomes clear that NoSQL backends will not *replace* traditional relational technologies. Each data model has its own benefits and drawbacks, and is most suitable for specific use cases. This may encourage developers to build *hybrid* data-intensive systems, also called *polystores* [8, 23, 24]. Such systems rely on a combination of multiple databases of different models, relational or NoSQL, each one chosen for its best features.

However, NoSQL database modeling is not yet as stable and mature as standard relational database design. In particular, NoSQL data representation does not often rely on a unique explicit schema. Even within the same paradigm, translating conceptual schema elements into physical data structures can be done in various different ways, depending on the anticipated usage of the data. Existing design techniques are either based on best practices [1–4] or target single data models [18, 21]. Some authors proposed to unify the NoSQL data models into a generic modeling framework [6]. However, since NoSQL design choices may greatly impact performance [9], it is important that the designer keeps full control on how data is stored physically, which is not always possible with a

generic data model. Furthermore, there is still a lack of models, methods, and tools supporting the design and querying of hybrid polystores, where relational and NoSQL databases are used in combination.

As a step to fill this gap, we introduce HyDRa (Hybrid Data Representation and Access), an integrated framework for conceptual modeling and manipulation of hybrid polystores. We focus on the HyDRa modeling language, as depicted in Figure 1, allowing one to specify the conceptual schema of a polystore, according to the *Entity-Relationship* model, as well as the physical schema of each of its underlying databases. HyDRa currently supports the relational model and the four most popular NoSQL data models, i.e., document, key-value, graph and column-based representations. The language enables the definition of mapping rules, to express correspondences between the conceptual schema elements and their physical database counterparts. Those rules enable possibly complex physical design choices, such as *data structure split*, *data instance partitioning*, *data heterogeneity* and *data duplication*.

The remaining of the paper is structured as follows. Section 2 summarizes the state-of-the-art approaches for NoSQL and hybrid polystore design. Section 3 presents and illustrates the HyDRa modeling language. Section 4 gives concluding remarks and anticipates future work.

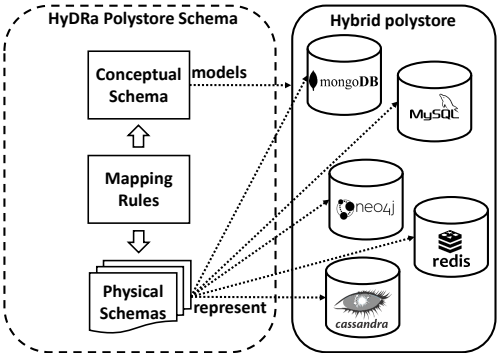

Fig. 1. Overview of the HyDRa modeling languages

2 Related Work

Database design for NoSQL applications is still an emerging research area. Current state-of-the-art approaches mainly consist of technology, data model-specific [18, 20, 21] design recommandations or best practices [1–4]. Roy-Hubara et al [22] made a systematic literature review on NoSQL database design. The SOS platform [6] provides a common interface to multiple NoSQL systems. It relies on a generic data model and provides automatic translations towards native backend implementations. BigDAWG [12] is a polystore implementation focusing on query optimization and data placement. NoAM [9] proposes a uniform

way to design NoSQL systems by abstracting the common features of each data model and by designing an aggregate identification step. Cabibbo [10] develops a JPA-based object mapper for multiple NoSQL backends, that enables the different data representation strategies presented in [9]. Herrero et al. [17] define a 3-step top-down design method from a conceptual model to physical structures. The logical schema is based on the data types specified in the conceptual schema, or on the build graph of dependent entities. The logical schemas are automatically proposed with optimizing performance as main objective. Bjeladinovic [8] presents an approach to design hybrid SQL/NoSQL databases. Based on database measures and requirements, the user is directed towards either a relational database design or to the NoAM approach.

The TyphonML model [7], which partly inspired HyDRa, also supports conceptual modeling of hybrid polystores, but imposes implicit restrictions on the way conceptual elements are physically translated in each different native backend. In other words, TyphonML does not leave developers the freedom to explicitly define the mappings between conceptual and physical schema elements of the polystore, which is at the core of HyDRa.

The approaches discussed above are either (1) design methods for particular data models, (2) abstraction-based approaches to conceptual modeling of NoSQL systems, or (3) polystore modeling approaches with limited control over the conceptual-to-physical mappings and no support to express cross-database overlapping within the polystore. In this paper, we propose an approach to hybrid polystore modeling that (1) provides users with a full and fine-grained control over the mapping between the conceptual schema and the underlying physical data structures, and (2) supports overlapping between the polystore databases.

3 The HyDRa Polystore Modeling Language

HyDRa polystore model language is composed of five main parts, each having its specific purpose. Figure 2 provides an abstract syntax of those main parts. *Conceptual schema* specifies the domain-specific data model of the complete system. *Physical schema* describes the data structures in the underlying physical databases. *Mapping rules* is where the mapping rules between conceptual schema elements and physical schema elements are expressed. *Databases* declares the physical databases and their respective configurations.

$$
\begin{aligned}
\langle \text{Schema} \rangle \models\ & \langle \text{ConceptualSchema} \rangle \langle \text{PhysicalSchema*} \rangle \\
& \langle \text{Mapping Rules} \rangle \langle \text{Databases} \rangle \\
\langle \text{PhysicalSchema} \rangle \models\ & \langle \text{RelationalSchema} \rangle \mid \langle \text{DocumentSchema} \rangle \mid \\
& \langle \text{KeyValueSchema} \rangle \mid \langle \text{GraphSchema} \rangle \mid \langle \text{ColumnSchema} \rangle
\end{aligned}
$$

Fig. 2. Abstract syntax of HyDRa language main components

The remaining of this section illustrates the HyDRa language, based on the example polystore schema of Figure 3. This schema, based on the *IMDB* dataset[1], involves three database back-ends: a key-value, a document and a relational database.

3.1 Conceptual schema

The *conceptual schema* represents the entities the polystore manipulates. As in standard database engineering methods, during conceptual design, the user specifies the domain model [11] based on *Entity-Relationship* model constructions. The domain is described by means of *entity types, attributes, binary relationship types, conceptual identifiers, n-ary relationship types* or *relationship types with attributes*. The conceptual schema of our *IMDB* example is declared in lines 1-36 in Figure 3. Entity types have attributes and declare one of several identifier(s) in their *identifier* section. Next we specify the relationship types and the roles played by the entity types within them. Relationship types can be *binary* or *n-ary*, and can also have attributes.

3.2 Physical schemas

The *physical schema* section of our model lets the designer specify how data is actually persisted in native databases. We support the relational data model as well as the four most popular NoSQL data models [16], namely document, key-value, column wide and graph-based representations. One of the key advantage of the physical section is the ability to represent each design technique of each data model, by providing the designer with full control on physical data structures. In our running example, this section spreads from line 38 to line 98. Below, we illustrate the physical data models supported by HyDRa, and how common design strategies fit in this language.

As *Physical Schemas* may represent five different types of data models, we had to define common terms across them to refer to data structures. Below we define the chosen terms of *Physical Structures*, *Physical Fields* as well as *References* allowing cross-database referencing.

A *Physical Field* is the term we use for data units in the corresponding technology-specific databases: *Columns* in relational, *Fields* in document, *Properties* in graph, *Columns* in column-oriented and *Value Properties* in key-value data models. For NoSQL complex data types such as arrays or objects fields, we use a different word by calling them *ComplexField*s.

A *Physical Structure* is an abstraction of technology-specific structures able to receive multiple data units. They contain multiple *Physical Fields*. Figure 4 describes the specification of this structure in our language. Typical structures include *Table* in a relational database, *Collection* in document database, and *Tablecolumn* in column oriented databases. For graph databases *Nodes* as well as *Edges* are considered physical structures. For key-value databases, we introduce

[1] `https://www.imdb.com/interfaces/`

```
conceptual schema cs{
  entity type Actor {
    id : string,
    fullName : string,
    yearOfBirth : int,
    yearOfDeath : int
    identifier { id }
  }
  entity type Director {
    id : string,
    firstName : string,
    lastName : string,
    yearOfBirth : int,
    yearOfDeath : int
    identifier { id }
  }
  entity type Movie {
    id : string,
    primaryTitle : string,
    originalTitle : string,
    isAdult : bool,
    startYear : int,
    runtimeMinutes: int,
    averageRating : string,
    numVotes : int
    identifier { id }
  }
  relationship type movieDirector{
    directed_movie[0-N]: Movie,
    director[0-N] : Director
  }
  relationship type movieActor{
    character[0-N]: Actor,
    movie[0-N] : Movie
  }
}

physical schemas {

  key value schema movieRedis :
     myredis {
    kvpairs movieKV {
      key : "movie:"[id],
      value : attr hash{
        title,
        originalTitle,
        isAdult,
        startYear,
        runtimeMinutes
      }
    }
  }

  document schema IMDB_Mongo : mymongo
     , mymongo2{

    collection actorCollection {
      fields {
        id,
        name:[fullname],
        birthyear,
        deathyear,
        movies[0-N]{
          id,
          title,
          rating[1]{
            rate: [rate] "/10" ,
            numberofvotes
          }
        }
      }
    }
  }

  relational schema myRelSchema {

    table directorTable{
      columns{
        id,
        fullname:
          [firstname]" "[lastname],
        birth,
        death
      }
    }

    table directed {
      columns{
        director_id,
        movie_id
      }

      references {
        directed_by : director_id ->
     directorTable.id
        has_directed : movie_id ->
     movieRedis.movieKV.id
        movie_info : movie_id ->
     IMDB_Mongo.actorCollection.movies.
     id
      }
    }
  }
}


  mapping rules{
  cs.Actor(id,fullName,yearOfBirth,yearOfDeath) -> IMDB_Mongo.actorCollection
     (id,fullname,birthyear,deathyear) ,
  cs.movieActor.character-> IMDB_Mongo.actorCollection.movies() ,
  cs.Director(id,firstName,lastName, yearOfBirth,yearOfDeath) -> myRelSchema.
     directorTable(id,firstname,lastname,birth,death),
  cs.movieDirector.director -> myRelSchema.directed.directed_by,
  cs.movieDirector.directed_movie -> myRelSchema.directed.has_directed  ,
  cs.movieDirector.directed_movie -> myRelSchema.directed.movie_info ,
  cs.Movie(id, primaryTitle) -> IMDB_Mongo.actorCollection.movies(id,title) ,
  cs.Movie(averageRating,numVotes) -> IMDB_Mongo.actorCollection.movies.
     rating(rate,numberofvotes) ,
  cs.Movie(id) -> movieRedis.movieKV(id) ,
  cs.Movie(primaryTitle,originalTitle,isAdult,startYear,runtimeMinutes) ->
     movieRedis.movieKV.attr(title,originalTitle,isAdult,startYear,
     runtimeMinutes)
}
```

Fig. 3. Example of a HyDRa schema, conceptual, physical and mapping rules section.

the *KeyValuePair* concept. It reflects a set of key-value pairs sharing the same pattern of keys and values (see lines 40-51 of Figure 3).

A *Reference* block expresses a link between two physical fields of physical structures. In a polystore, a source field could reference a target field declared in a different database, and relying on a different data model. Therefore HyDRa offers the possibility to express cross-references between heterogeneous databases.

For instance, lines 91-95 in Figure 3 declare three references. Reference *directed_by* indicates that physical field *director_id* values are also stored in the *id* field of *directedTable*. This reference is the expression of a foreign key of the many-to-many table *directed*. The other column of this join table, *movie_id*, is a multiple hybrid reference, as *has_directed* targets *id* in the *movieRedis* key-value database and *movie_info* targets document database *IMDB_Mongo*.

Relational schemas Relational schemas are composed of tables and columns. Columns may only contain simple values. Lines 73-97 show the declaration of a relational schema structures. Following tables declaration are the references declaration. Source fields of the declared references are part of the current relational schema, however target fields may be in a different structure.

Document schemas Document schemas follow a JSON-like data model. It consists of key-value pairs organized by documents, each document field may in turn be a document, allowing embedded structures at certain levels of depth. Available design methods for document databases are described by MongoDB [1], the leading technology for this data model. Embedding data structures is referred to as *one to few*. Lines 53-71 show a document database schema declaration. Lines 61-68 show how we can declare such a nested structure, using complex field *movies* as an array. If, for design purposes or for technical reasons, embedding documents is not possible, the user can choose to use referencing values across collections of documents, this is referred to *one to many* design. Our model allows such constructions using reference blocks, as described above.

Key-Value schemas Key-value schemas simply consist of key-value pairs, with no constructs allowing references between data instances. Querying data in this model is done only using *put* and *get* operations on the key part. This apparent simplicity may lead to possibly complex schema design problems when deciding how to organize the data. One needs to carefully design and manage the chosen key patterns. Design methods [5, 21] and best practices [4] identified two main patterns. The first one, called *Key value per field*, creates a key-value pair for each atomic field. The key is composed of different elements identifying a particular atomic instance. Examples of key patterns for this design includes *ENTITY:[identifier]:FIELD*. It results in data such as *MOVIE:tt0118715:TITLE* as key, and a binary object *The Big Lebowski* as value. The second design type, *Key Value per object*, uses complex data types instead of simple atomic value, the value contains now multiple fields. This allows the grouping of multiple fields under the same key. Lines 40-51 illustrate this pattern.

Column-oriented schemas Column-oriented schemas rely on row identifiers (*rowkey*), and each row is composed of groups of key-value pairs (*column families*). Design methods identified in [2,3], such as *Row per object representation* or *Single cell per object* are also supported in the HyDRa language. We refer to the HyDRa companion website [15] for an illustrative example of a column-based schema.

Graph schemas Graph schemas represent data as *Property graphs*. The data model is composed of *Nodes* and *Edges* that may contain *Properties*. The common way to design graph databases is described by the leading technology of graph databases, Neo4j [19]. *Nodes* usually represent entities and relationships between data are expressed using *edges*. Again we refer to our companion website [15] for an illustrative example.

3.3 Mapping Rules

The mapping rules section of an HyDRa polystore schema specifies links between the conceptual schema elements and the physical structures. Exploiting the possibly hybrid nature of those mapping rules, the designer can specify complex constructions such as *data structure split*, *data instance partitioning*, *data heterogeneity* and *data duplication* (see Section 3.5).

Figure 4 exposes the abstract syntax of mapping rules types and of their components. The left-hand side of the rule (before the arrow) is the *conceptual* component and the right-hand side corresponds to the *physical* component. Two types of mapping rules are supported: *Entity mapping rules* and *Role mapping rules*. Mapping rules are in lines 100-111 in the polystore schema of Figure 3.

⟨PhysicalStructure⟩ ⊨ ⟨Table⟩ | ⟨Collection⟩ | ⟨TableColumn⟩ |
⟨Node⟩ | ⟨Edge⟩ | ⟨KeyValuePair⟩
⟨EntityMappingRule⟩ ⊨ ⟨EntityType⟩(⟨Attribute+⟩)→
⟨PhysicalStructure⟩(⟨PhysicalField+⟩)
⟨RoleMappingRule⟩ ⊨ ⟨Role⟩→⟨Reference⟩ | ⟨ComplexField⟩

Fig. 4. Abstract syntax of HyDRa structures and mapping rules

EntityMappingRule is a type of rule used to map *Conceptual Entity types* to *Physical Structures*. A conceptual entity type can be mapped to one or more heterogeneous structures. We provide some examples of mapping rules in Figure 3. First, at line 101, entity type *Actor* and its attributes are mapped to collection *actorCollection* in document database schema *IMDB_Mongo* (schema at lines 53-71). Second, at line 103, entity type *Director* is mapped to table *directorTable*, belonging to relational database schema *myRelSchema* (lines 73-97). Last, entity

type *Movie* is mapped to three physical structures and one complex field (lines 107,108, 109, 110). Line 108 maps attributes *averageRating* and *numVotes* to physical fields contained into a third-level embedded structure *rating* in the *movies* array of *actorCollection*.

RoleMappingRule is another mapping rule type that maps *Role*s of *Relationship type*s to *Reference* blocks or to *ComplexField*s. Lines 102, 105 and 106 are examples of such mapping rules for relationship types *movieDirector* and *movieActor*.

3.4 Physical Databases

The physical database section is used to declare the actual databases linked to the physical schemas, and provide their connection information. Each physical schema can be linked to one or several declared database(s). Figure 5 shows the databases declared for the physical schemas of Figure 3.

```
1  databases {
2    mariadb mydb {
3      host: "localhost"
4      port: 3307
5      dbname : "mydb"
6    }
7    redis myredis {
8      host:"localhost"
9      port:6379
10   }
11     mongodb mymongo{
12       host : "localhost"
13       port: 27100
14     }
15     mongodb mymongo2{
16       host : "localhost"
17       port: 27000
18     }
19   }
```

Fig. 5. Databases declaration section

3.5 Benefits of HyDRa

Data duplication & heterogeneity HyDRa allows data duplication at the level of conceptual objects as well as at the physical schema level. Data duplication at the level of entity types can be expressed through multiple entity mapping rules, with the same entity type as left-hand side, but mapping it to several different physical structures. An example was given above with the mappings of attribute *primaryTitle* of *Movie* entity type (lines 107 and 110) which is mapped to both a document database and a key-value database. HyDRa also allows one to duplicate an entire physical schema into several databases. For instance, line 53 in Figure 3 declares that physical schema *IMDB_Mongo* is stored in both databases *mymongo* and *mymongo2*.

Composed fields The physical fields of an HyDRa schema can have complex values. This is made possible by means of complex physical field declarations and related mapping rules. For instance, line 79 of Figure 3 specifies that the value of column *fullname* in relational table *directorTable* results from the concatenation

of conceptual attributes *firstname* and *lastname*. This is expressed using the entity mapping rule at line 103. As another example, physical field *rate* at line 65 concatenates the *rate* conceptual attribute value with the "/10" string constant.

Data structure split Conceptual entity types can be split and stored in multiple and heterogeneous databases. Multiple entity mapping rules can be expressed for distinct fragments of a single entity type, e.g., by splitting its attributes into multiple and possibly heterogeneous databases. For instance, conceptual entity type *Movie* is composed of eight attributes, but those attributes are stored either in the *IMDB_Mongo* schema or in *movieRedis* schema, or in both physical schemas. As expressed by the mapping rules of lines 107, 108, 109 and 110, some of the movie attributes are subject to data duplication across several physical schemas, while attributes *isAdult, startYear, runtimeMinutes* are only present in the *movieRedis* schema.

Data instance partitioning Using data instance partitioning, an HyDRa polystore schema can map only a *subset* of the data instances of a given entity type to a particular physical structure. The data instances are discriminated based on user-defined conditions on the value of a particular entity type attribute. For instance, in Listing 1.1, a mapping rule expresses that the instances of entity type *Movie* that have an *averageRating* value greater than 9 must be stored in the *topMovies* physical structure.

```
1     cs.Movie(id,primaryTitle,averageRating,numVotes) -(averageRating > 9)->
       IMDB_Mongo.topMovies(id,title,rate,numberofvotes),
```

Listing 1.1. Mapping rule data instance partitioning

4 Conclusion

This paper introduces HyDRa, a conceptual framework for hybrid polystore modeling and manipulation. It focuses on the HyDRa modeling language, able to *conceptually design hybrid polystores* while preserving the possibility to design data at physical level and exploit the strengths of each native data model. The conceptual and physical abstraction levels are linked together through a set of *mapping rules*, allowing complex features such as *data structure and instance overlapping* across heterogenous databases. The use of HyDRa is supported by an Eclipse plugin, publicly available on GitHub [15]. This plugin includes a textual editor as well as a conceptual data access API generator.

The HyDRa framework can be used, among others, to build mediated architecture from pre-existing inter-related databases, to assist in a database reverse engineering context or to express explicit schema for schemaless databases. Polystore data management still faces various open challenges for the research community. In particular, specifying polystore schemas and mapping rules still remains a manual task. As future work, we aim at developing automated schema inference and mapping rules recommendation approaches. Our current research

agenda also includes extension of generation of conceptual data manipulation APIs, the automation of schema evolution, data migration, and query adaptation in hybrid polystores [13, 14].

Acknowledgements. This research is supported by the F.R.S.-FNRS and FWO via the EOS project 30446992 SECO-ASSIST.

References

1. 6 rules of thumb for mongodb schema design. `https://bit.ly/3gYTh8y`
2. Cassandra data modeling best practices. `https://bit.ly/3eeYGGY`
3. Hbase schema case study. `https://bit.ly/3nX52y5`
4. Spring data redis - retwis-j. `https://bit.ly/33hEFcg`
5. Atzeni, P., Bugiotti, F., Cabibbo, L., Torlone, R.: Data modeling in the NoSQL world. Computer Standards & Interfaces **67**, 103149 (2020)
6. Atzeni, P., Bugiotti, F., Rossi, L.: Uniform access to non-relational database systems: The sos platform. In: CAiSE. pp. 160–174. Springer (2012)
7. Basciani, F., Di Rocco, J., Di Ruscio, D., Pierantonio, A., Iovino, L.: Typhonml: a modeling environment to develop hybrid polystores. In: MoDELS (2020)
8. Bjeladinovic, S.: A fresh approach for hybrid sql/nosql database design based on data structuredness. Enterprise Information Systems **12**(8-9), 1202–1220 (2018)
9. Bugiotti, F., Cabibbo, L., Atzeni, P., Torlone, R.: Database design for nosql systems. In: ER. pp. 223–231. Springer (2014)
10. Cabibbo, L.: ONDM: an object-NoSQL datastore mapper. Faculty of Engineering, Roma Tre University (2013)
11. Carlo, B., Ceri, S., Sham, N.: Conceptual Database Design: An Entity-Relationship Approach. Benjamin/Cummings (1992)
12. Duggan, J., Elmore, A.J., Stonebraker, M., Balazinska, M., Howe, B., Kepner, J., Madden, S., Maier, D., Mattson, T., Zdonik, S.: The BigDAWG polystore system. ACM Sigmod Record **44**(2), 11–16 (2015)
13. Fink, J., Gobert, M., Cleve, A.: Adapting queries to database schema changes in hybrid polystores. In: IEEE SCAM. pp. 127–131 (2020)
14. Gobert, M.: Schema evolution in hybrid database systems. In: VLDB PhD Workshop (2020)
15. Gobert, M.: HyDRa repository (2021), `https://github.com/gobertm/HyDRa`
16. Hecht, R., Jablonski, S.: Nosql evaluation: A use case oriented survey. In: 2011 International Conference on Cloud and Service Computing. pp. 336–341
17. Herrero, V., Abelló, A., Romero, O.: Nosql design for analytical workloads: variability matters. In: ER. pp. 50–64. Springer (2016)
18. de Lima, C., dos Santos Mello, R.: A workload-driven logical design approach for nosql document databases. In: iiWAS. pp. 1–10 (2015)
19. Neo4j: Modeling designs. `https://neo4j.com/developer/modeling-designs/`
20. Pokornỳ, J.: Conceptual and database modelling of graph databases. In: IDEAS16
21. Rossel, G., Manna, A., et al.: A modeling methodology for nosql key-value databases. Database Syst. J **8**(2), 12–18 (2017)
22. Roy-Hubara, N., Sturm, A.: Design methods for the new database era: a systematic literature review. Software and Systems Modeling **19**(2), 297–312 (2020)
23. Sadalage, P.J., Fowler, M.: NoSQL distilled: a brief guide to the emerging world of polyglot persistence. Pearson Education (2013)
24. Schaarschmidt, M., Gessert, F., Ritter, N.: Towards automated polyglot persistence. Datenbanksysteme für Business, Technologie und Web (BTW 2015) (2015)